SecurePot: システムコールフックを利用した安全なソフトウェア実行系

SecurePot: Secure Software Execution System Based on System Call Hooks

大山恵弘 † 加藤和彦 †,††
Yoshihiro OYAMA Kazuhiko KATO
† 科学技術振興事業団 さきがけ研究 21
PRESTO, JST
†† 筑波大学 電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba
{yosh,kato}@osss.is.tsukuba.ac.jp

実行を許すシステムコールを制限して既存のバイナリコードを安全に実行するためのシステム SecurePot を開発した。SecurePot はコードが発行するシステムコールの呼び出しをフックし、設定に応じて、そのコードの実行を強制終了したりユーザに実行継続の可否を問う等の処理を行う。従来のシステムと異なり、SecurePot は実行を許す呼び出しを設定するファイルの作成を容易にする機構を持つ。それはインクリメンタルかつ対話的にファイルを作成できる機構と、GUI 上でファイルを作成できる機構を含む。

1 はじめに

得体の知れないコードの実行はまさに冒険である。しかし世界には得体の知れない（従って悪意があるかもしれない）コードがあふれており、悪意がないことを保証できないコードを実行したいことが少なくない。

悪意があるかもしれないコードを安全に実行するのに有効なのは、コードが行える操作（例えばファイルを作る操作）を制限できるシステムである。このようなシステムを以降では *sandbox* システムと呼ぶ。

我々が開発している SecurePot は sandbox システムの一つである。SecurePot はユーザが与えたセキュリティポリシー（以降で適宜単にポリシーと呼ぶ）に基づき、アプリケーションが発行するシステムコールをフックし、システムコールの種別や引数に応じて実行を継続したり強制終了したりする。SecurePot はユーザレベルで実現されている。

本稿は sandbox システムの使い勝手を大きく高める機構を提案する。それはユーザへの問い合わせを利用してポリシーをインクリメンタルに作成する機構と、GUI 上でポリシーを作成する機構からなる。

2 目標と解決すべき問題

本研究の長期的目標は、セキュリティポリシーの作成開始から完成までの時間と労力の極小化である。

従来の sandbox システムではポリシーの作成に多くの時間と労力がかかる。従来のシステムでは、まず、ユーザは極力安全なポリシーを記述し、そのポリシーでアプリケーションを実行する。ポリシーが厳しすぎて実行を完了できない場合、安全だと納得できる範囲でポリシーを緩いものに書き換え、再び最初からアプリケーションを実行する。アプリケーションが実行を完了できるまでポリシー書き換え作業は続く。これは面倒な工程である。

ユーザがアプリケーションの挙動を正確に予測して、最初から完全な（すなわち安全かつ実行を完了できる）ポリシーを書ければ、何度もポリシーを書き換える必要はないが、それは非現実的である。多くのアプリケーションは複雑であり、その実行中に呼び出されるシステムコールや読み書きされるファイルを実行前に把握できない[1]。ただし、アプリケーションの最初の実行に、ユーザの意図に極力沿ったポリシーを与えることは、後に発生するポリシーの書き換えを少なくする点で意味がある。

上の考察を踏まえ我々が取り組んだ問題は、(1) ポリシーを書き換えて完全なものにする作業の簡単化と、(2) アプリケーションの最初の実行に与えるための、ユーザの意図に極力沿ったポリシーを簡単に作れるツールの構築である。SecurePot の設計方針の

[1] truss や strace を使えばかなり正確に挙動を把握できるが、コードに悪意があるかもしれない場合その方式はとれない。

根幹は、それらの問題の解決にある。

3 SecurePot の概要

3.1 システムの構成

SecurePot システムの構成を図 1 に示す。悪意があるかもしれないアプリケーションと OS の間に SecurePot は位置する。ユーザはセキュリティポリシーを記述したファイル（ポリシーファイル）を通じて各システムコールをどう実行するかの指示を SecurePot に与える。SecureFurnace はポリシーファイルを作るためのツールである（後述）。

SecurePot はカーネルモジュールやライブラリではなく、単体のアプリケーションである。SecurePot はコマンドライン引数にポリシーファイル、アプリケーションのコードと引数を受け取り、そのアプリケーションをそのポリシーファイルに書かれたポリシーに従って実行する。例えば、

```
% securepot tar.plc tar xvf foo.tar
```

を実行すると、SecurePot は tar.plc というポリシーファイルに従って tar を実行する。

3.2 セキュリティポリシー

セキュリティポリシーは、フックされるシステムコールの種別と引数、システムコールがフックされた後に SecurePot がとる反応を指示する[2]。セキュリティポリシーの例を図 2 に示す。

ポリシーファイルにおける各行の最初の語は、以降の語が示すシステムコールの呼び出しへの反応を指示している。以下のどれかを最初の語に書ける。

ignore そのシステムコール呼び出しの実行を許可する（フックすらしない）

deny そのシステムコール呼び出しが呼び出されたらアプリケーションの実行を強制終了する

allow-and-print そのシステムコール呼び出しの種別と引数の情報を表示して実行を継続する

ask そのシステムコール呼び出しの実行の許可不許可をユーザに問い合わせる

各行の二番目の語はシステムコールの種別を指示する。全てのシステムコールを意味する all という特殊なシステムコール種別を使ってもよい。

[2]SecurePot は非システムコール関数の実行も制御できる。その話題については付録 A に述べる。

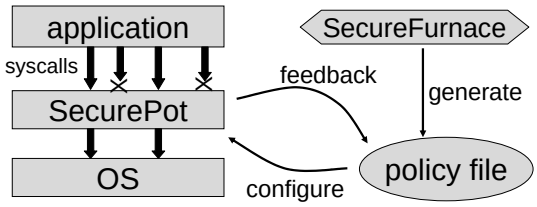

図 1: SecurePot の構成

各行の三番目以降の語は、（もしその指示が必要ならば）システムコールの引数の値を指示する。例えば

```
ignore open 1 "/dev/zero"
```

は、第一引数が /dev/zero であるような open の実行を許可するよう指示している。

特殊なシステムコール種別 path を用いると、読み書きを許すファイルパスの制限ができる。例えば、

```
deny path rw /etc/*
```

と書くと、/etc ディレクトリに存在する全ファイルの読み出しと書き込みが禁じられる。

特殊なシステムコール種別 sock を用いると、ソケットによる接続を制御する IP アドレスとポート番号を指示できる。例えば、

```
deny sock connect 123.45.67.89 80
```

と書くと、IP アドレス 123.45.67.89 を持つホストの 80 番ポートへの connect 操作が禁じられる。

3.3 セキュリティポリシーのインクリメンタルな作成

SecurePot はシステムコールの実行の可否をユーザに問い合わせるたびにその答を記憶する。そして、アプリケーションの実行終了後に、ポリシーファイルをその答を反映したものに自動的に更新する。

この機構によって、ポリシーファイルを対話的にインクリメンタルに作ることが可能になる。例えば、最初は「全てのシステムコール呼び出しでユーザに問い合わせる」という単純なポリシーを与えてアプリケーションを実行する。そして、呼び出されたシステムコールを問い合わせを通じて一つ一つ実行許可していく。すると、実行終了後、そのポリシーファイルは、それらのシステムコールの実行を許し、そうでないものの実行を問い合わせるものに変わる。

ユーザの意図を細かい単位で反映させるために、一部の問い合わせは複数項目に渡って行われる。例え

```
ask all
ignore open 1 "/dev/zero" 2 O_RDONLY
ignore open 1 "/usr/lib/locale/ja/ja.so.1" 2 O_RDONLY
ignore open 1 "/usr/lib/locale/ja/methods_ja.so.1" 2 O_RDONLY
ignore open 1 "/usr/lib/locale/ja/LC_MESSAGES/SUNW_OST_OSCMD.mo" 2 O_RDONLY
ignore open 1 "/usr/lib/locale/ja/LC_TIME/SUNW_OST_OSCMD.mo" 2 O_RDONLY
ignore ioctl 2 TCGETA
```

```
ignore brk
ignore write
ignore close
ignore fstat
ignore llseek
ignore memcntl
ignore mmap
ignore munmap
```

図 2: echo の実行に使うセキュリティポリシー

ば、open システムコールの問い合わせでは、実行を許可する場合、全ての open の実行を許可するのか、現在の引数のファイルに限り open の実行を許可するのかが問われる。また、ファイルパスの読み書きの制御に関する問い合わせでは、現在アクセスしようとしているファイルのみを制御するか、そのファイルと同じディレクトリにある全ファイルを制御するか、そのファイルがあるディレクトリ以下の階層にある全ファイルを制御するかが問われる。

問い合わせに際しては、実行状態に関する詳しい情報を表示し、ユーザの判断を支援する。例えば open システムコールの問い合わせではカレントディレクトリが表示される。

3.4 セキュリティポリシー作成のための GUI

我々はポリシー作成 GUI ツール SecureFurnace を提供し、ポリシー作成を一層容易にする。現在 SecureFurnace としては web の form と CGI を使ったプロトタイプが動いている（図 3）。ユーザは SecureFurnace 上でメニューから項目を選んだりチェックボックスをクリックするだけでポリシーファイルを作成できる。例えば、“can create/update files” というチェックボックスを有効にすると、open と creat システムコールの実行を許可するようなポリシーファイルが生成される。

ユーザがシステムコールなどの低レベルの仕様を極力意識せずポリシーを作れることを SecureFurnace は目指している。無論、ポリシー作成ツールが GUI であることは高レベルな水準でポリシーを書けることの必要条件でも十分条件でもない。我々が SecureFurnace を通じて得たい学術的知見は、GUI ベースのポリシー作成ツールが、高レベルな水準でのポリシー記述に、どのように/どの程度貢献するかである。

現在のプロトタイプはポリシーファイルの新規作成はできるが、読み込みと編集はできない。将来は SecureFurnace をブラウザを使わない単体のアプリケーションとして開発し、ポリシーファイルの読み込みと編集ができるようにしたい。

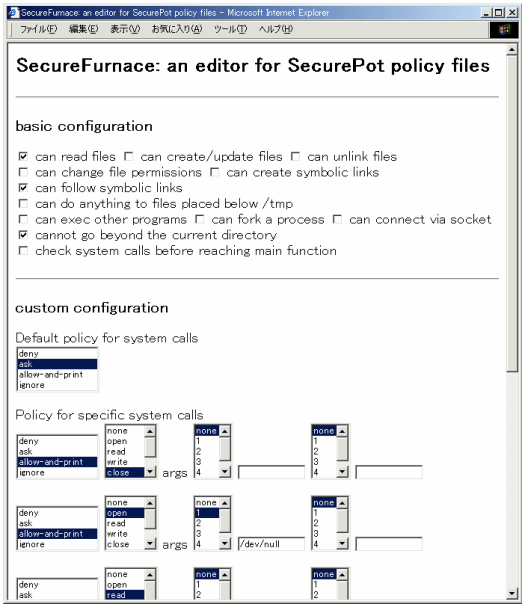

図 3: SecureFurnace のプロトタイプ

4 実装

我々は SecurePot を Solaris 上に実装した。SecurePot は起動されると子プロセスを作り、子プロセス上で引数に受け取ったアプリケーションを走らせる。親プロセスは子プロセスのシステムコール呼び出しにフックを入れ、子プロセスがそのフックで停止するたびに適切な処理を行う。フックの実装には/proc ファイルシステムを利用した。親プロセスは、/proc 以下にあるファイルに制御データを書き込む操作を通じ、望みのシステムコールの実行前に子プロセスを停止させる。同様の操作を通じ、親プロセスは停止した子プロセスの実行を再開したり、子プロセスのメモリ空間を参照してシステムコールの引数の情報を得たりする。

5 関連研究

SecurePot に似たシステムに、Janus [4]、MAPbox [1]、ASL [6] がある。これらのシステムはポリシーを対話的に作成する機構を持たない。MAPbox はアプリケーションの分類ごとにポリシーの雛型を

用意してポリシー記述の手間を省いているが、分類内の全アプリケーションが雛型に沿った挙動を示すわけではないという問題を持つ。

FMAC tools [5] は、悪意がない入力でアプリケーションを実行してファイルアクセスの挙動を記録し、その挙動を強制するセキュリティポリシーを生成する。悪意があるかもしれない入力でのそのアプリケーションの実行で、そのポリシーが使われる。我々の方法は、彼らの方法と異なり、悪意があるかもしれないアプリケーションの実行中に出される問い合わせへの答からポリシーを作る。また、彼らの方法はパスの制限のみを扱うが、我々の方法はより一般化されていて、様々な操作の制限を扱える。

SubDomain [2] は、既存アプリケーションを対象に、読み書き実行の各操作を行えるパスを制限する sandbox システムである。SubDomain はパス以外の制限（例えば通信の制限）ができず、かつ、ポリシーの記述を支援する機構も持たない。

Java2 では、SecurePot と同じく宣言的なポリシーを使って Java プログラムにファイル操作の制限や通信の制限を課すことができる。Java2 Runtime Environment には GUI 上でポリシーを作成、編集できるツール policytool が含まれている。今後我々は policytool の長所を取り入れる形で SecureFurnace を改良したい。

システムコールのラッパを記述して操作的な形でシステムコールの実行を制御する研究が多数存在する [3]。これらは SecurePot の宣言的なポリシー記述のアプローチよりも融通がきく反面、記述が面倒である。二つのアプローチには利点と欠点があり、片方が常によいとは言えない。

6 結辞と今後の課題

Sandbox システムの使い勝手を高める機構として、インクリメンタルで対話的なポリシー作成機構と GUI 上でのポリシー作成機構を提案した。それらが SecurePot 以外の sandbox システムでも採用され、それらの使い勝手が高まることを我々は望む。

現在測定されている SecurePot のオーバヘッドは最も小さい場合でも 52%ある。これは他のシステム [1, 4, 6] の数値よりかなり大きい。それらのシステムと SecurePot の間には実装方式において大きな相異はないので、そのオーバヘッドは SecurePot の現実装の未熟さによるものと考えている。

今後は、オーバヘッドの縮小、SecureFurnace の非プロトタイプ版の実装、使い勝手をさらに向上させる新しい機構の考案などに取り組みたい。

SecurePot は
`http://www.osss.is.tsukuba.ac.jp/~yosh/securepot/` から取得できる。

参考文献

[1] Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[2] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious Server Security. In *Proceedings of the 14th Systems Administration Conference (LISA 2000)*, December 2000.

[3] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 2–16, May 1999.

[4] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th USENIX Security Symposium*, July 1996.

[5] Vessilis Prevelakis. Sandboxing Applications. In *Proceedings of 2001 USENIX Annual Technical Conference, FREENIX Track*, June 2001.

[6] R. Sekar and T. Bowen and M. Segal. On Preventing Intrusions by Process Behavior Monitoring. In *Proceedings of the 1st Workshop on Intrusion Detection and Network Monitoring*, April 1999.

A 非システムコール関数の実行制御

SecurePot では libc などのライブラリ関数やユーザが定義した一般の関数も実行制御できる。例えば、fprintf の実行前に問い合わせをさせることができる。実行許可した関数の実行中はシステムコールおよび関数のフックを一時的に無効にするよう設定できる（有効のままにもできる）。この機構により、ユーザはより大きい単位で実行制御するポリシーを書けるとともに、フックの回数を減らして性能を向上させる可能性が生まれる。関数の実行中にフックを一時的に無効にする設定は、ユーザがその関数のコードを信頼できる場合に有用である。例えば、自分が書いたスクリプト処理系で、信頼できないスクリプトを解釈実行する場合に有用である。関数の実行制御は、フックしたい関数のコードの先頭にブレークポイントを入れることにより実装されている。